Kuiper: Enterprise Cloud Plug-and-Play Network Management Platform

Kaiyang Zhao[1], Han Zhang[2], Tianpeng Zhang[3], Yantu Tong[2], Zeji Xiao[2], Xuefeng Liu[2], Xia Yin[1], Jianping Wu[2]
[1]*Department of Computer Science and Technology, Tsinghua University*
[2]*Institute for Network Sciences and Cyberspace, Tsinghua University*
[3]*Yunshan Networks*

Abstract

Enterprise cloud networks are becoming critical infrastructure, posing new challenges to management platforms. The rapid iterative development of microservice applications in these networks brings frequent deployment demands. Meanwhile, heterogeneity in the underlying physical network makes adaptation more difficult. However, existing solutions either do not support flexible networking for microservices or cannot adapt quickly to heterogeneous networks. To address these challenges, we propose Kuiper, a plug-and-play enterprise cloud network management platform that meets application requirements and adapts to heterogeneous networks. Kuiper provides a unified network model for expressing microservice networking requirements across heterogeneous clouds. Building on this model, Kuiper enables multi-vendor configuration generation using large language models through configlet graphs and document-driven iterative reasoning. Kuiper also supports runtime configuration verification, enabling precise localization of configuration anomalies. Compared to state-of-the-art frameworks, Kuiper can save hours of network deployment work for application releases and shorten the upgrade time for new vendor equipment to just a few hours. Kuiper has been deployed in more than 20 companies and used by hundreds of enterprise network administrators.

1 Introduction

Enterprise cloud networks are becoming the infrastructure of choice. Many enterprises are migrating workloads from public clouds back to on-premise data centers [12]. Recent surveys [1] indicate that 53% of enterprises prioritize private clouds for new workload deployments over the next three years. Meanwhile, their services are transitioning from monolithic structures to distributed microservices frameworks. The evolution of enterprise cloud networks stems from a fundamental objective: achieving a *plug-and-play* experience comparable to that of public clouds.

The management platform must integrate new devices within hours (*plug*) and automate configurations within minutes (*play*). However, existing solutions often assume greenfield deployments rather than the incremental growth of enterprise systems [13, 15]. To address these goals, state-of-the-art solutions [3, 10, 18–20, 22, 27, 33, 34, 36–40] leverage Natural Language Processing (NLP) to parse configuration manuals for rapid device adaptation. Other approaches [2, 3, 5, 8, 9, 14, 17, 21, 23–26, 28] attempt to satisfy requirements via centralized network configuration. Existing systems can be categorized into standard protocol-based systems and non-standard protocol-based systems according to their underlying device interaction mechanisms. They remain inadequate in the following three key aspects.

Standard protocol-based systems [2, 5, 9, 11, 14, 17, 25] struggle to provide fast multi-vendor adaptability. These systems are predicated on the assumption that devices expose uniform interfaces for configuration. This assumption is violated in practice, where protocol support is often incomplete. This discrepancy forces continuous device-specific adaptation, thereby limiting deployment agility.

Non-standard protocol-based systems [3, 21, 28] still fail to provide accurate plug-and-play control for enterprise clouds. These systems are predicated on the assumption that controller-side translation can reliably bridge high-level control intents and vendor-specific device operations. However, this assumption becomes increasingly fragile in enterprise-cloud deployments, where automation must translate larger volumes of complex commands while supporting the rapid integration of newly introduced vendor devices. Moreover, these systems are not designed to address the realities of enterprise clouds, which are characterized by heterogeneous resource pools and fast-evolving microservice applications.

Both standard protocol-based and non-standard protocol-based systems focus on configuration generation and delivery, while overlooking runtime configuration consistency verification. They cannot detect configuration drift introduced during execution, leaving enterprise clouds vulnerable to inconsistent and unsafe operating states.

Kuiper framework. To address these challenges, we propose Kuiper, a plug-and-play management platform for enter-

prise cloud networks (§ 3). Kuiper improves the management paradigm through three core designs:

Unified network model (§ 3.1). Existing approaches realize the data plane in hosts through framework-specific mechanisms, such as IP-based [4, 6, 16, 29], tunnel-based [32], or routing-based forwarding [31], and cross-site deployments introduce additional forwarding plugins. This host-centric design fragments the network semantics of one application across heterogeneous environments. It becomes problematic when an application spans Kubernetes, OpenStack, VMware, or geographically distributed resource pools.

To address this problem, Kuiper introduces *eVPC* (enterprise Virtual Private Cloud), a VPC abstraction for applications deployed across heterogeneous enterprise infrastructure. Each eVPC represents one application's isolated logical network. At the data plane, Kuiper realizes VRFs on edge network devices rather than in host networking stacks, thereby enabling near line-rate forwarding in the network. At the control plane, programmable service templates map stable eVPC abstractions to concrete device-side services through validation, rendering, and incremental reconciliation. This design enables consistent network realization across heterogeneous enterprise clouds.

Multi-vendor configuration generation (§ 3.2). Existing multi-vendor provisioning [21,28] still relies heavily on experts to handcraft translation rules, and recent generation-based approaches [2, 3, 5, 8, 9, 14, 17, 21, 23–26, 28] often become inaccurate when configuration logic is complex.

The key observation is that configuration generation should operate on small semantic units with explicit relations. Based on this idea, Kuiper builds a configlet graph, where each configlet is a minimal configuration unit and each edge captures either hierarchy or dependency. Kuiper then performs document-driven iterative reasoning: it retrieves vendor evidence for the relevant configlets, generates local vendor-specific templates, and expands the required subgraph in dependency order. This design reduces expert effort, avoids long-context errors, and enables low-cost support for new vendors.

Runtime configuration verification (§ 3.3). Existing configuration checking schemes either rely on structured device state exported through model-driven interfaces, or use tree-based abstractions [21] to organize controller-side configuration dependencies. However, these approaches do not verify whether the actual running command-line interface (CLI) on devices still semantically matches controller intent after deployment. Kuiper therefore introduces a tree-based verifier for semantic consistency checking. It converts device running CLI into a hierarchy-aware search tree and decomposes controller intent into structured search expressions. It then matches the two representations level by level, using order-aware alignment to distinguish among missing, extra, conflicting, and wrongly scoped commands. As a result, Kuiper supports verification for CLI-centric devices and localizes configuration drift at command granularity.

Contributions. Kuiper has been deployed in production for over 6 years, used by hundreds of network administrators across 20 large-scale environments, including data centers, ISPs, and enterprise intranets. Before 2024, Kuiper already supported devices from 8 vendors. To address growing vendor fragmentation, we integrated a large language model (LLM) module, which has since enabled support for 8 additional vendors. Empirical results show that Kuiper reduces the onboarding time for a new vendor from two weeks to a few hours. Separately, user feedback indicates that Kuiper improves operational efficiency. We summarize our primary contributions as follows:

1. We conduct an extensive study of enterprise cloud networks and present new insights into the critical requirements for network management (§ 2).
2. We design Kuiper, a plug-and-play network management platform tailored for enterprise cloud networks. Kuiper features a unified network model alongside a multi-vendor adaptation module. It further enables runtime configuration consistency verification (§ 3).
3. We evaluate Kuiper in production environments (§ 4). The results show that Kuiper saves hours of application deployment time and weeks of development effort for integrating new devices, while enabling efficient configuration consistency verification.

2 Background and Motivation

This section first introduces enterprise cloud network controllers in production and the new requirements they face (§ 2.1). We then present the motivation for Kuiper (§ 2.2).

2.1 Enterprise Cloud Controllers in Practice

Enterprise cloud controllers face different design requirements compared to public clouds, as Table 1 shows. Public cloud controllers typically provide standardized virtual overlay networks on uniform large-scale topologies using in-house hardware and proprietary protocols. In contrast, enterprise controllers must support customized business scenarios across complex and variable topologies. To avoid vendor lock-in, they orchestrate both underlay and overlay networks by pushing configurations directly to multi-vendor physical devices. Tenant isolation allows public cloud controllers to recover by re-instantiating a tenant's configuration, whereas enterprise controllers require targeted repairs to avoid disrupting existing network services. Overall, enterprise clouds differ from public clouds in service scenarios, hardware adaptation, and orchestration workflows, leading to distinct requirements for controllers.

Microservice architectures amplify this operational complexity by decomposing applications and generating massive inter-service network policies. In Figure 1, our survey of 18

Table 1: Comparison of enterprise and public cloud controllers

Controller	Service Scenarios			Hardware Adaptation		Orchestration Workflow		
	User	Target	Operation Object	Device Type	Interaction Protocol	Validation Goal	Fault Strategy	Criteria
Enterprise Cloud	Admin	Underlay & Overlay	Physical Network	Multi-vendor	Standard	Configuration Consistency	Atomic Rollback	Compliance
Public Cloud	Tenant	Overlay	Virtual Network	In-house	Proprietary	Declaration Verification	Re-instantiation	Concurrency

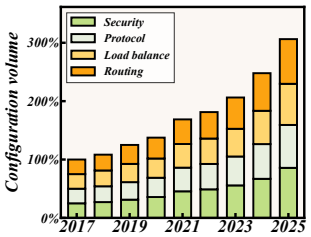


Figure 1: Config analysis.

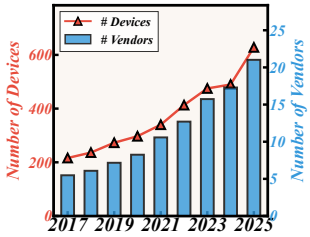


Figure 2: Added devices.

enterprises shows that configuration operations grow by over 20% annually. Among them, routing, load balancing, protocol state management, and access control account for more than 90% of all change requests.

These differences introduce three new requirements for enterprise cloud controllers.

Requirement 1: High-performance Network Orchestration for Microservices. The need for high-performance network orchestration in microservice scenarios appears at both the control plane and the data plane. At the control plane, enterprise clouds usually rely on VLANs, VRFs, and similar mechanisms to achieve logical isolation, which makes operations and management complex. This requires the control plane to support efficient orchestration and automated configuration delivery. In the data plane, the overhead of isolation technologies such as encapsulation also needs to be reduced.

Requirement 2: Fast Multi-vendor Device Integration. The scale of enterprise infrastructure continues to grow. The number of newly deployed network devices increases by about 15% each year, as shown in Figure 2. At the same time, multi-vendor procurement is adopted because of supply-chain security considerations. The rate of introducing new device vendors is about 20% per year. Devices from different vendors differ in configuration syntax, interface models, and control protocols, which makes management of new devices challenging. YANG is a data modeling language for describing network device configuration and operational state. Although YANG aims to provide standardized device management interfaces, in practice vendors differ in YANG model coverage. Therefore, multi-vendor device integration must abandon the assumption of complete YANG interfaces and instead rely on the CLI supported by all devices.

Requirement 3: Fast Configuration Drift Localization. When resources on network devices are exhausted, management sessions such as gRPC or SSH may terminate. By that point, the device may already have executed part of the configuration commands, leaving partial configurations on the device.

Table 2: A summary of the features of existing systems for network controllers.

Support	Method	Systems	R1	R2	R3
Standard Protocol	Flow-driven	Orion [9], Andromeda [5], Onix [17], Avenir [2]	◐	○	●
	Model-driven	OpenDayLight [25], Intent [11], Lumi [14]	◐	◐	●
Non-standard Protocol	Manual	Robotron [28], POSEIDON [21]	◐	◐	●
	Automatic	NAssim [3]	◐	◐	○
		Our (Kuiper)	●	●	●

* ● indicates that the requirement is fully satisfied, ◐ indicates partial satisfaction, and ○ indicates that the requirement is not satisfied.

In addition, when legacy devices are brought under management, newly issued configurations can conflict with residual configurations. Finally, during fault repair, administrators may execute commands for manual adjustment, causing inconsistency between device state and the configuration stored in the controller. Each lifecycle event of microservices can trigger network configuration changes. Unlike public clouds, which can destroy and rebuild virtual network components, enterprise clouds rely on atomic rollback to recover inconsistent configurations after configuration drift occurs. Therefore, the controller must precisely localize drift.

2.2 Motivating Kuiper

Unfortunately, existing cloud network controllers are falling behind the times, especially when it comes to satisfying the aforementioned new requirements. As shown in Table 2, we classify existing systems into two types based on their underlying protocol support, standard protocol and non-standard protocol, and characterize their limitations.

Standard protocol-based systems. These systems interact with network devices using unified and predefined interfaces. Depending on their abstraction level, these systems can be further categorized as flow-driven or model-driven. The flow-driven systems [2, 5, 9, 17] use protocols like OpenFlow to directly program the data plane, whereas the model-driven systems [11, 14, 25] use YANG models or similar abstractions to provide configuration abstractions. Although standard protocol systems offer structured management capabilities, they struggle with massive vendor heterogeneity. Flow-driven systems perform well in highly controlled environments but fail to manage diverse physical devices. Many commercial routers and switches completely lack OpenFlow support. Model-

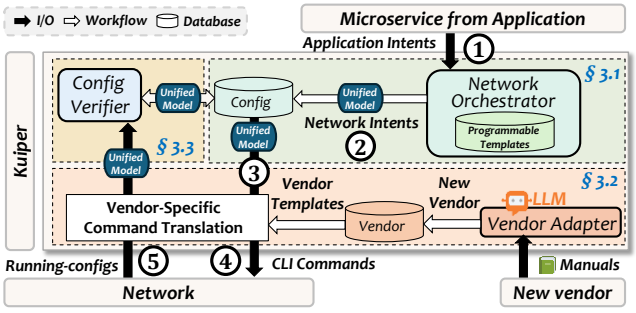


Figure 3: Kuiper overview.

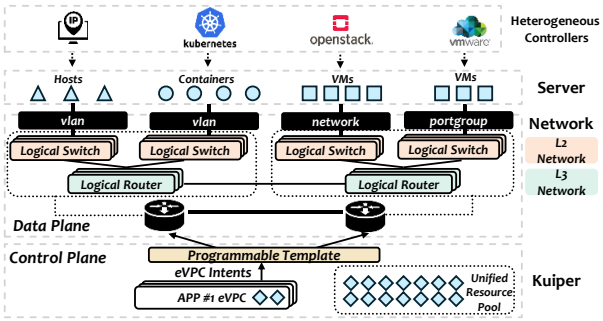


Figure 4: eVPC abstracts heterogeneous host-side environments and realizes application networks through edge-device configuration.

driven systems attempt to solve this by abstracting device configurations. However, different equipment vendors inevitably implement heavily customized YANG extensions. This severe model fragmentation destroys the promise of out-of-the-box multi-vendor adaptation. It forces developers to continuously write custom patches for specific vendor models. This continuous manual effort severely limits deployment agility.

Non-standard protocol-based systems. In order to manage heterogeneous devices lacking standard protocol support, controllers must interact through the command line interface. Depending on whether the vendor adaptation is performed automatically, these systems are further divided into manual and automatic approaches. The manual approaches [21, 28] require network operators to write specific translation scripts and parsing rules for every vendor device. This massive manual effort is error-prone and imposes a significant burden on users and developers. The automatic approaches [3] attempt to automate syntax translation and state parsing. These systems, however, are unable to simultaneously provide deployment agility and transactional safety. First, command line interactions inherently lack transactional properties. Second, current automatic wrappers fail to guarantee strict configuration consistency. They struggle to execute atomic rollbacks during network deployment failures. These structural blind spots leave the enterprise network in an unsafe partial state.

3 Kuiper Design

This section presents the design of Kuiper. At the center of Kuiper is a unified network model. After an administrator submits application requirements (①), Kuiper expands them through programmable service templates into vendor-agnostic network intents and writes the intents into the configuration database (②). It then uses vendor adaptation templates to compile unified intents into executable CLI (③,④), which allows new vendors to be added by introducing only the corresponding templates. After deployment, Kuiper continuously reads device running configurations (⑤) and compares them with target intents at the unified model. This enables Kuiper to verify consistency after deployment or during periodic checks and to localize configuration drift.

3.1 Unified Network Model

Existing enterprise cloud networks are usually managed separately by framework (e.g., Kubernetes, OpenStack, or VMware). This management model is centered on underlying device deployment rather than applications. Once an application spans multiple frameworks or adopts a multi-site redundancy strategy, its network configuration is split across multiple independent management units. Kuiper aims to provide a unified network model for enterprise clouds. The connectivity, isolation, and service-chaining requirements of the same application should be expressed in one model.

Kuiper proposes eVPC as the network model for microservice applications in enterprise clouds, as shown in Figure 4. Each application has one eVPC. For example, in a Kubernetes setting, Kuiper maps a single Namespace to a single eVPC, so that the network boundary matches the application boundary. It realizes consistent data-plane encapsulation in the network. For implementation, it uses programmable service templates to provide a flexible control plane.

eVPC components. eVPC consists of three types of logical components. The first is the Logical Switch (LS). An LS represents a logical Layer-2 broadcast domain, and endpoints connected to the same LS share the same Layer-2 connectivity semantics. Kuiper does not require these endpoints to come from the same resource type. One LS can connect Pods, virtual machines, and bare-metal servers at the same time. The second is the Logical Router (LR). An LR provides Layer-3 interconnection among multiple LSs within the same eVPC and serves as the aggregation point for internal policy control. The third is virtualized network functions, including the virtual firewall (vFW), virtual load balancer (vLB), and virtual gateway (vGW). Therefore, all policies can be expressed in one logical graph.

Existing approaches realize the data plane on hosts through different frameworks and their associated plugins. Cross-site communication further introduces different forwarding plugins. Based on the data plane type, existing frameworks can be divided into three types, as shown in Figure 5. IP-based frameworks [6, 16, 29] place applications and the physical net-

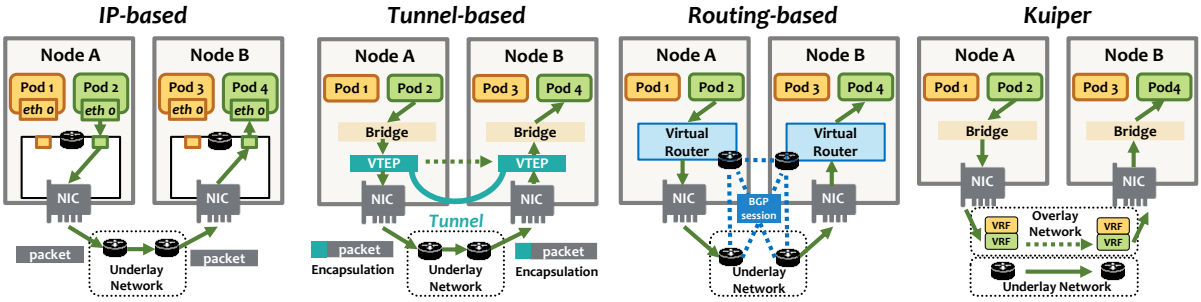


Figure 5: Kuiper eVPC data plane solution: VRF-based L2/L3 network.

work in the same IP address space, offering high performance but relying on the underlying infrastructure. Tunnel-based frameworks [32] encapsulate packets using a tunneling protocol, which introduces additional overhead. Routing-based frameworks [31] implement a virtual router in the host kernel and use BGP to route packets, but they incur high operational complexity.

eVPC data plane. Kuiper unifies the network semantics that were previously scattered across Kubernetes Namespaces, OpenStack tenant networks and VLANs into eVPC, and realizes them consistently in the network. It no longer relies on the host network stack of each framework, but instead places the implementation on edge network devices. Specifically, LS and LR are realized as MAC-VRFs and IP-VRFs, respectively, on edge network devices such as leaf switches. Once traffic leaves the host, both forwarding and encapsulation are handled in the network.

eVPC control plane. Network services and backend APIs in enterprise clouds keep evolving. Kuiper does not encode these changes into the controller system. Instead, it pushes them into programmable templates. The controller system maintains only LSs, LRs, and network functions, while templates are responsible for the concrete implementation. For each type of logical network representation, the template executes three stages in sequence: validation, rendering, and incremental reconciliation. The validation stage checks whether preconditions such as interface resources have been satisfied. The rendering stage converts logical objects into metadata and execution specifications. The former records object identifiers and dependencies, and the latter provides the parameters. The incremental reconciliation stage compares the target state with the current state and generates updates only for changed fields.

Listing 1 uses the construction of the Bridge Domain corresponding to an LS to illustrate the template-based instantiation workflow above. The template first verifies that dependent resources are ready, then renders the logical properties of the LS into underlying configuration objects, and in the update stage performs precise modifications only on fields that differ, such as the gateway. By replacing controller code with programmable templates as the unit of control plane evolution, Kuiper confines the scope of changes for adding new network services, adapting to new device capabilities, and integrating

Listing 1: Compact bridge domain logic.

```
function buildBridgeDomain(vxlan, ap) {
  const device = ap.spec.device;
  const name = `${device}/${ap.spec.l2vpn}`;
  // 1) Resource-gated allocation
  const alloc = pool(device, "bridge-domain").allocate(name);
  if (!alloc) return null;
  // 2) Metadata & Dependencies
  const deps = { vxlan: vxlan.name, accessPoint: { l2vpn: vxlan.
      name, device } };
  const labels = { l2vpn: vxlan.name, device };
  const l3vpn = queryL3vpn(vxlan.name);
  if (l3vpn) labels.l3vpn = l3vpn;
  // 3) Service spec construction
  const spec = { l2vpn: vxlan.name, device, vni: vxlan.spec.vni };
  if (vxlan.spec.gateway) spec.gateway = vxlan.spec.gateway;
  return Entity({
    meta: { type: "bridge-domain", name, allocation: alloc, depends:
        deps, labels },
    spec
  });
}
// Incremental reconciliation
function reconcileBridgeDomain(ap, vxlan) {
  if (!vxlan) return buildBridgeDomain(acquireVxlan(ap.spec.l2vpn)
      , ap);
  const desiredGw = vxlan.spec.gateway;
  return listBridgeDomains({ vxlan: vxlan.name }).map((bd) => {
    if (desiredGw) bd.spec.gateway = desiredGw;
    else delete bd.spec.gateway;
    return bd;
  });
}
```

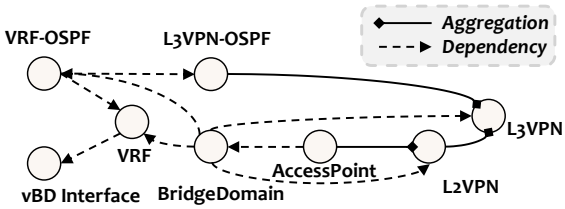


Figure 6: Example of resource dependency handling.

new API backends to the template layer, without intrusive modification to the controller core.

Network configurations often involve complex dependencies among resources. As shown in Figure 6, enabling OSPF on an L3VPN is not a local change, but a modification that may affect other dependent objects. For example, when OSPF is enabled on an existing L3VPN, the corresponding VRF-OSPF objects associated with its downstream VRFs must also be created or updated accordingly. To handle such dependencies, Kuiper explicitly models the relations among resources and propagates changes from the updated resource to all affected dependent resources.

Configlet-Graph Prompt

```
...
You project one planned configlet node into target-vendor CLI.
      Output template text only.
...
The target-vendor PDF evidence is authoritative.
Cross-vendor local blocks only express intent and ownership
      boundaries.
Keep one consistent target-vendor family for this node.
...
First line must be this node's own entry command when the node
      opens a view.
Do not emit child configlet entry commands.
Keep indentation relative to this node.
...
```

Iterative-Reasoning Prompt

```
...
Self-review pass:
- Re-check the previous draft against the updated evidence.
- If the updated evidence supports a better target-vendor
      hierarchy or command family, revise the draft.
- Continue retrieval and correction only around local issues.
- Keep already-valid parts unchanged.
...
```

Figure 7: Key prompt excerpts in Kuiper.

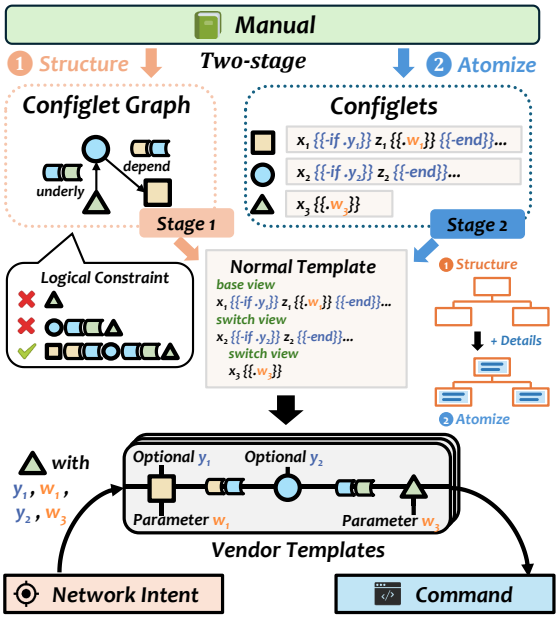


Figure 8: Vendor adaptation. The manual provides evidence. Kuiper performs structure extraction to build the configlet graph and configlet atomization to obtain minimal CLI units.

3.2 Multi-vendor Configuration Generation

eVPC provides a vendor-neutral unified network model. The next step is to translate this model into vendor-specific configurations. The challenge is whether, when the network introduces devices from a previously unsupported vendor, the system can achieve this goal using only the vendor's public configuration documents without experts. Kuiper aims to support translation from the unified network model to executable vendor-specific CLI within hours when a new device is introduced.

In our experience, the previous approach of manually writing rules by experts takes from two weeks to several months. Different vendors often realize the same network behavior with different keywords and parameters. This makes the cost of hand-writing translation logic grow multiplicatively with the number of vendors and behaviors. Moreover, CLI validity depends on the current configuration view and the order of command execution. It must explicitly reason about the execution partial order among configuration objects. Finally, public configuration manuals often span hundreds of pages and contain much irrelevant material, so they are not suitable for direct automated analysis.

Kuiper adopts a two-stage approach that separates global configuration logic from local reasoning. The *structure* stage builds a configlet graph to make configuration relationships and execution constraints explicit. The *atomize* stage breaks configuration details into small semantic units, allowing the model to process each with a focused context.

Configlet graph. For a specific device type, we construct a directed graph $G = (V, E)$, where $E = E_{\text{struct}} \cup E_{\text{dep}}$. Each vertex $v \in V$ is called a configlet, and a configlet is the smallest indivisible configuration semantic unit for a given device type. It serves as a uniform abstraction for configuration generation. The graph contains two types of directed edges, namely structural edges E_{struct} and dependency edges E_{dep}. For E_{struct}, if the configlet A underlies a subitem B, a structural edge is added from A to B, and this edge indicates that B is semantically subordinate to A. As a result, structural edges capture configuration hierarchies and view nesting. For E_{dep}, these edges represent prerequisite conditions and execution order constraints. When the correct application of configlet B depends on configlet A already existing or being executed first, a dependency edge is added from A to B. Therefore, edges in the graph explicitly encode command ordering dependencies and cross-module resource constraints.

Order and dependencies no longer rely on the model's implicit memory over long contexts. They are guaranteed explicitly by the configlet graph. At the same time, the system avoids placing the entire device manual into one context, which reduces attention degradation and error propagation in long-context generation. In addition, when facing a new vendor, Kuiper needs to revise only a small number of edge relations rather than rebuilding the entire graph.

Document-driven iterative reasoning. Kuiper first extracts structural relations and then retrieves from the target vendor manual the syntax and command evidence. Based on these constraints, the model first generates a candidate configuration. The system then identifies parts that are missing, conflicting, or unsupported by the documents, and continues retrieval and correction only around these local issues. This makes use of the prior knowledge, but that knowledge cannot determine the final output. A structure or command is kept only if it is supported by the target document. Each round uses the errors from the previous round to shrink the search space, and uses

Controller (intended)

```
interface Gi0/0/1
  ip address 10.0.1.1/24
  mtu 1500
  ip ospf cost 100
  no shutdown

interface Gi0/0/2
  ip address 10.0.2.1/24
  no shutdown

interface Gi0/0/3          Missing
 ip address 10.0.3.1/24
 no shutdown

router bgp 65001
  neighbor 10.0.1.2
    route-map POLICY in
```

Device (running)

```
interface Gi0/0/2    Reordered (OK)
  ip address 10.0.2.1/24
  ip access-group ACL_IN in  Extra
  no shutdown

interface Gi0/0/1    Reordered (OK)
  ip address 10.0.1.1/24
  mtu 9000                   Conflict
  ! extra comment            Ignored
  ip ospf cost 100
  no shutdown

router bgp 65001
  neighbor 10.0.1.2

route-map POLICY in    Wrong Scope
```

Figure 9: Controller intent vs. device running-config. Legend: Missing (entity not applied); Extra (rogue config); Conflict (value mismatch); Wrong Scope (wrong hierarchy); Ignored (comments or blank lines). Note: Block reordering (e.g., interfaces) is safely resolved without functional impact.

external documentary evidence.

The design above is implemented through a prompt-based workflow. Figure 7 shows how Kuiper realizes the preceding design with prompts. We added these instructions over time based on experience to ensure that the model follows our design. Figure 8 shows the system workflow. The system first retrieves the relevant sections from documentation for the functional domains involved in the input intent, and performs LLM-based inference to recover configuration objects and their relations, yielding a configlet graph. The system then performs projection independently for each configuration unit in the graph, generating a vendor-specific parameterized template. At runtime, the system activates the required configuration units, extracts the corresponding subgraph and instantiates the templates to produce the final CLI commands.

Generated templates undergo functional integration tests before deployment. If a test fails, engineers use the two-stage outputs to revise the configlet graph or CLI commands. Keeping logic and commands separate simplifies correction, which typically takes a few hours of manual work.

3.3 Runtime Configuration Verification

After translating the unified model into device-specific CLI commands and deploying them, Kuiper still needs to verify whether the current running configuration is consistent with the intent generated by the controller. Existing systems usually assume that devices can return YANG-modeled structured state through interfaces such as NETCONF and RESTCONF, so they can compare two structured configuration views. However, in our experience, this assumption often fails. A substantial fraction of devices in enterprise clouds still provide only CLI-based management interfaces. In addition, a device API may return success even when the configuration transaction has not completed, which creates silent drift.

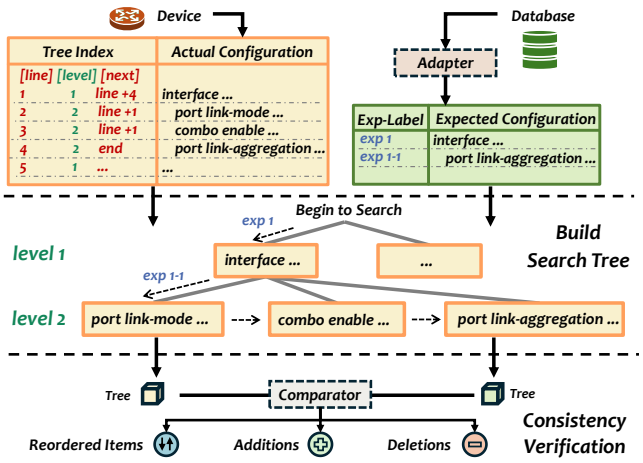


Figure 10: Consistency verification.

Existing configuration verifiers usually compare two structured states and assume that devices can return a directly parsable structured configuration view. But in enterprise clouds, the device-side input comes from vendor-specific raw configuration byte streams collected via `show running-config`. Kuiper must verify whether the actual running configuration on the device constitutes a valid implementation of the controller intent. If not, the system must pinpoint which CLI fragments violate the intent constraints.

Challenge. Configuration commands have order and dependency relations, as shown in Figure 9. Consistency checking does not ask whether two configurations are line-by-line identical. It asks whether the device-side configuration is semantically consistent with the controller-side configuration. Order-insensitive differences, such as a different interface order, as well as variations like configuration comments, should not be reported. By contrast, missing, conflicting, extra, and wrongly nested configurations must be reported. Methods based on text diffs or hashes misclassify some semantically consistent cases as inconsistent. The challenge is how to identify hierarchy and order in configuration commands.

Kuiper addresses this challenge with a CLI-centric verifier. It converts controller-side structured intent and device-side raw CLI into comparable representations. This design both covers devices that support only CLI and produces command-level discrepancy reports. It has been noted that the number of indents signifies the relationship between configuration instructions. Thus, we create a search tree to represent the current configuration and transform the association into a hierarchical relationship in the tree.

Tree-based search. As shown in Figure 10, the consistency verification process consists of four stages. First, the system retrieves the actual configuration from the device and constructs a search tree indexed by the configuration structure. Second, it fetches the expected configuration from the database and decomposes it hierarchically into search expressions. Third, these expressions are matched against the search tree level by

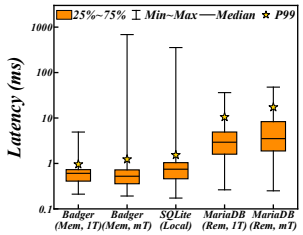


Figure 11: Intent rendering latency.

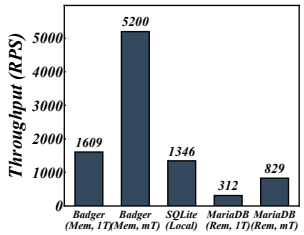


Figure 12: Intent rendering throughput.

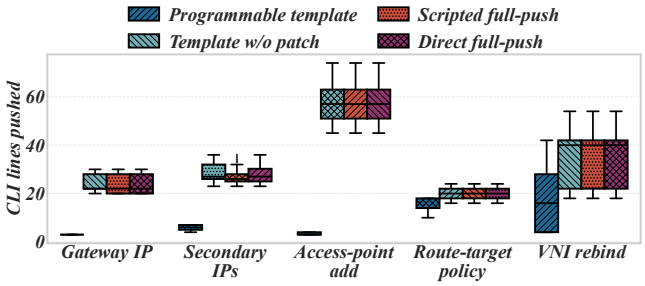


Figure 13: Device-side command churn.

level. The result is considered valid only if all sub-expressions match successfully. Finally, the search results are processed to identify discrepancies.

Node alignment. Within a matched scope, the system aligns sibling nodes according to whether order is semantically meaningful. For order-sensitive rule sequences, it uses a two-pointer scan over the ordered child lists to distinguish missing, redundant, and modified rules. For order-insensitive items, it builds keyed indices and aligns nodes set-wise rather than by position. To separate benign reordering from real insertions and deletions, the system further computes the longest common subsequence (LCS) over candidate sequences: nodes present on both sides but outside the common subsequence are treated as reorderings, while nodes present on only one side are treated as insertions or deletions. Reordering is ignored for order-insensitive items, but retained as a command-level anomaly for order-sensitive items.

4 Kuiper in Production

In this section, we evaluate Kuiper through production deployments and experiments. Kuiper has been deployed in production across 20 enterprises, including private clouds and ISPs. Our main results are as follows:

- Kuiper improves efficiency in both the database and the control plane (§ 4.1).
- Kuiper improves configuration generation by achieving 97.63% F1 and reducing manual template effort (§ 4.2).
- Kuiper improves verification effectiveness by achieving 100% F1 within the evaluated scope and sustaining a verification throughput of 3,000 items/s (§ 4.3).
- Kuiper demonstrates production operability through two real-world deployment cases: an enterprise cloud and an ISP network (§ 4.4).

4.1 Kuiper eVPC Efficiency

Database performance across database backends. The deployment consists of physical servers with E5-2630 v4 CPUs, virtual machines with Intel Broadwell CPUs, and H3C CR16000 routers. We utilize Badger as the in-memory database, SQLite for local testing, and MariaDB for remote database services. We evaluate database latency on both physical and virtual servers. Figure 11 illustrates the P99 latency results for Kuiper using Badger (in-memory), SQLite (local), and MariaDB (remote). 1T and mT denote the single-thread and multi-thread settings, respectively. The results show that all database configurations satisfy the latency requirements (experience value based on customer feedback, within the millisecond range). Notably, while the multi-thread performance of the in-memory database can become a bottleneck in extreme scenarios, its performance remains comparable to the local test database.

Additionally, we measure the throughput in terms of responses per second for different database types. As shown in Figure 12, the distributed multi-thread configuration achieves superior performance. The choice of deployment scheme for Kuiper ultimately depends on the specific scale and requirements of the enterprise cloud network.

Table 3: Model sensitivity of Kuiper.

Model	Overall Recall	Overall F1
DeepSeek-V3.2	89.16%	89.06%
GLM-5	87.84%	87.74%
Qwen3-Coder	98.86%	97.63%

Programmable template performance across update strategies. Meanwhile, we evaluate whether incremental reconciliation reduces device-side updates. We compare the full *programmable template* with three variants: *template w/o patch*, which keeps validation and rendering but disables reconciliation, *scripted full-push*, and *direct full-push*. Figure 13 shows that only the full template reduces the number of CLI lines pushed. This confirms that rendering logical objects into device-specific specifications is not enough. The main reduction comes from comparing target and current states and updating only changed commands.

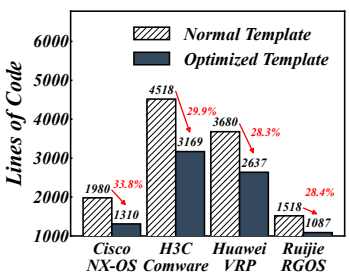


Figure 14: Template scale.

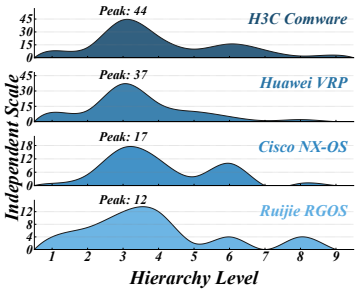


Figure 15: Template layering.

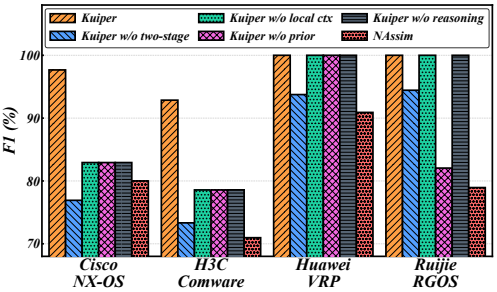


Figure 16: Accuracy with different vendors.

4.2 Configuration Generation Accuracy

Template efficiency across vendors. To establish a ground truth dataset for evaluating generation accuracy, we manually developed templates for 8 different vendors. From this effort, we curated the evaluation dataset by selecting four vendors (Cisco-NXOS-7, H3C-Comware-v7, Huawei-VRP-8, and Ruijie-RGOS-11) with the most comprehensive and functionally similar feature sets.

Our evaluation reveals two major efficiency gains. First, Kuiper templates reduce the total template size (lines of code) by approximately 30% compared to traditional raw templates, as shown in Figure 14. Second, regarding human effort, Kuiper optimizes the manual workload from two weeks to just a few hours. These benefits become even more pronounced as the complexity of network configuration increases, as Kuiper allows engineers to bypass redundant work.

Regarding logic complexity, the Kuiper vendor templates across the four devices form a *configlet graph* where the longest path reaches a depth of 9 in Figure 15, reflecting the inherent complexity of CLI commands. Most configlets consist of three layers. However, this is not a universal constant but a result of the high frequency of VRF operations in our enterprise cloud networks. Furthermore, we observed significant variance in nesting depth across vendors. For instance, Ruijie RGOS has vendor-specific command structures at command level 8.

Generation accuracy across vendors. The candidate models are Qwen3-Coder-480B-A35B-Instruct, DeepSeek-V3.2, and GLM-5. We define strict command-level metrics. Command-level recall measures the fraction of required ground-truth commands successfully generated. Command-level precision measures the fraction of generated commands that are correct and functional. We report the F1 score, $2PR/(P+R)$, which combines precision P and recall R to measure quality. Based on the model comparison, in Table 3, we use Qwen3-Coder-480B-A35B-Instruct as the default model for the remaining experiments.

We compare Kuiper with an adapted NAssim [3] baseline that keeps NAssim's framework, but replaces its original BERT mapper with the same LLM used by Kuiper. As shown in Figure 16, Kuiper consistently outperforms this baseline. The two-stage design is critical. Averaged over four vendors, Kuiper achieves 97.63% F1, while removing two-stage generation drops F1 to 84.61%. The remaining ablations show that each component is useful. Removing local context (local ctx), prior constraints (prior) and document-driven iterative reasoning (reasoning) all reduce performance.

Table 4: System performance and overhead.

Category	Metric	Value
Processing	Latency	0.4 s/item
	Consistency Verification	3,000 items/s
Overhead	CPU Utilization	<1%
	Memory Footprint	~30 MB
	Disk Usage	~12 MB

4.3 Verification Effectiveness

End-to-end deployment overhead. We measure the end-to-end overhead of Kuiper. In an experiment involving the deployment of 5,000 real-world network configurations, Kuiper completed the entire process in 33 minutes and 20 seconds. The consistency verification computation was completed in less than 2 seconds. On the server side, CPU utilization was negligible, and both memory and storage overheads remained minimal throughout the process, as shown in Table 4.

Verification accuracy across baselines. Our benchmark is a reproducible multi-vendor CLI consistency dataset built from real config features of four vendors, with 115 challenge cases designed to stress semantic equivalence, reordering, and scope ambiguity. These cases were constructed from historical inconsistency incidents and checked by network engineers, and the benchmark size is limited by the records. Our evaluation covers only the inconsistency types observed in these incidents. Since deployment, we have received no customer reports of verification errors.

Table 5 reports detection, localization, and deviation-type F1 scores, along with the false-positive rate on equivalent configurations. Kuiper achieves 100% F1 and produces no false positives on this benchmark. The two text baselines differ only in whether they normalize CLI lines before diffing. Although preprocessing improves detection and lowers false positives, both text baselines still fail to localize the responsible CLI commands. POSEIDON [21] detects some controller-side

Table 5: Verification performance.

Method	Detection F1 score	Localization F1 score	Type F1 score	False-Positive Rate
Kuiper	**100.0%**	**100.0%**	**100.0%**	**0.0%**
Kuiper w/o Preprocessing	86.5%	83.8%	92.5%	17.4%
Kuiper w/o LCS	94.2%	80.0%	93.4%	**0.0%**
Text-Diff	71.5%	0.0%	67.3%	44.3%
Preprocessed Text-Diff	89.5%	0.0%	80.6%	13.0%
POSEIDON [21]	83.1%	0.6%	0.0%	22.6%

object changes, but its results do not align with device-side CLI spans, which leads to near-zero localization accuracy and zero deviation-type accuracy. The ablation results indicate that preprocessing removes spurious mismatches from formatting artifacts, and that LCS-based alignment is required to separate command reordering from real additions or deletions. While this demonstrates high accuracy within the test scope, broader coverage remains future work.

4.4 Case Studies

We present two real-world cases from these enterprises.

Case 1: Elastic network orchestration in an enterprise cloud. A large enterprise operates a production cloud infrastructure spanning $O(10)$ data centers, each hosting distinct applications and $O(10)$ heterogeneous resource pools. As the enterprise migrates business workloads toward containerized platforms, the underlying network has grown to $O(1000)$ switches from multiple vendors, including H3C, Huawei, and Ruijie, and $O(10{,}000)$ servers. The system has been running in this environment for six years. The enterprise's core business traffic exhibits highly bursty traffic patterns. During peak periods, applications must rapidly expand from a primary data center to secondary sites, requiring network connectivity across resource pools and data centers.

Before deploying Kuiper, operators considered several conventional approaches but found them insufficient for this scale. A large L2 design places containers in a single flat broadcast domain, whose rigid topology cannot satisfy application-specific isolation and connectivity requirements. A centralized SDN design requires controllers to install fine-grained exact-match rules on physical switches, but the TCAM capacity of commodity enterprise switches cannot support $O(100{,}000)$ containers with second-level churn. A BGP-based host-routing design shifts the pressure to core switches. Network-wide BGP UPDATE flooding overloads core-switch CPUs, and routing convergence can exceed 30 seconds.

With the adoption of Kuiper, all resource pools achieve L2 and L3 interconnectivity, as shown in Figure 17. The vGW provides gateway functionality for cross-subnet east-west and north-south traffic. NAT is deployed on the firewall to publish various services to external networks.

The enterprise's network operations team comprises over 30 engineers, more than half of whom have more than ten years of experience. Before the cloud upgrade, the enterprise supported approximately 120 business applications. By 2025,

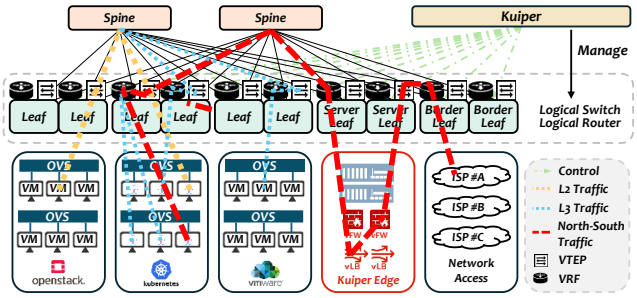


Figure 17: Multi-Resource pool access.

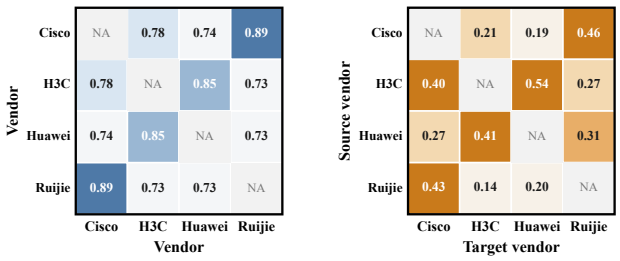
after the upgrade, its cloud infrastructure hosted approximately 500 applications (VPCs). In 2025, 90% of the 6,700 network operations tickets were processed automatically.

Case 2: Tenant-network provisioning in an ISP network. An ISP deploys Kuiper in its resource-access backbone, which spans $O(10)$ sites and supports $O(1000)$ heterogeneous application networks. Each application network is associated with pre-allocated resources for a different application. The backbone provides L2 and L3 access through SRv6, enabling high-performance inter-site connectivity over shared carrier infrastructure.

Before Kuiper, provisioning an application network required repeated manual coordination between application teams and network operators. Teams had to request VLAN allocation, VRF creation, tunnel setup, and routing-policy changes for each iteration of their virtual networks. This process was slow and error-prone, especially because multiple application teams shared the same backbone while requiring different isolation, connectivity, and routing semantics.

Template efficiency in production. In a 64-leaf-switch deployment, a single eVPC template invocation encapsulates substantial control-plane complexity: across five eVPC variants, each template encapsulates 18–26 API types and 32–1500 API calls. The composition is dominated by routing-related APIs, while security, load-balancing, and other functions are integrated into the same abstraction. This shift allows network engineers to reason about high-level architectural constraints instead of manually orchestrating tens of API types and, for switch-access scenarios, up to 1500 underlying API calls.

Operator reviews. With Kuiper, application teams view the network not as fragmented configuration snippets but as composable templates. The deployment workflow has evolved from manual negotiation to the submission of structured network descriptions. Based on operational feedback from ISP administrators, Kuiper accelerates provisioning and improves operational efficiency by 10× across the end-to-end network delivery cycle.



(a) Structural similarity across vendor pairs.

(b) Top-3 exact-path retrieval accuracy across vendor pairs.

Figure 18: Cross-vendor structural similarity and retrieval performance.

5 Experience & Lessons

5.1 LLM in Network Management

Kuiper explores the role of LLMs in enterprise cloud network management in production environments, yielding the following lessons at the current stage.

Why does Kuiper use CLI for LLM-based network configuration? We initially considered JSON/YAML as the representation for LLM-based network configuration, since YAML is human-friendly and JSON is widely used as a machine-readable interchange format. Initially, we asked the model to produce structured configuration objects and then translated them into device commands. This design worked well in small examples, but it soon became a practical bottleneck in production networks. The reason is that JSON/YAML requires an extra modeling layer, and this layer is often incomplete, vendor-specific, or unavailable for legacy devices. In contrast, CLI is the lowest common denominator across network devices. It is a semi-structured language consisting of commands, parameters, hierarchy, views, and operational context. Moreover, lightweight representations reduce token cost and latency [7].

How should we use the knowledge in LLMs? A natural design is to rely only on vendor manuals. For each task, the system retrieves documents and asks the LLM to generate commands from them. This design was insufficient in our production setting. Manuals are often fragmented and command-centric. They rarely capture view context, defaults, operator conventions, or interactions across commands. If we rely only on retrieval, we lose useful prior knowledge already stored in the model. However, the model's prior knowledge is not safe to trust directly. It mixes vendors, versions, and historical practices. A command pattern may be correct for one device, but wrong for another. Therefore, Kuiper treats the LLM's knowledge as a candidate source, not as the source of truth. The model first proposes likely command structures and configuration patterns. Then Kuiper uses manuals, device state, and validators to confirm syntax and semantics before deployment.

How does structure help cross-vendor transfer? Different vendors often implement the same network feature with a similar structural skeleton. This structural regularity provides a useful transfer signal even when the concrete command strings are not identical. As shown in Figure 18(a), the same feature frequently exhibits high structural similarity across vendors, indicating that one vendor's template can serve as a strong prior for another. At the same time, the signal is not sufficient for direct execution. Figure 18(b) uses a Top-3 retrieval metric. For each target feature, we rank source-vendor templates by structural similarity and count a hit if the correct implementation appears in the top three. This shows that structure is useful for cross-vendor retrieval. It narrows the search space to a few likely candidates, but the final template still requires confirmation before deployment.

In our earlier work as a network-product vendor, API-based adaptation to a new vendor took over two months. This included about one month of functional validation by one network engineer and one month of development by two software engineers. Incomplete API support made completion times unpredictable. CLI-based adaptation reduced this to about two weeks, with required functions generally available through CLI. Unlike API integration, which required substantial software-development expertise, CLI adaptation could be completed by a network engineer with basic programming skills. Dedicated software engineers were no longer needed for each adaptation, allowing network engineers to apply their expertise directly. LLM-assisted generation further automates the manual development work that previously took two weeks. Network engineers instead review critical steps and validate the generated templates before deployment.

5.2 Design of the Unified Controller

Vendor-controller orchestration and NETCONF/YANG both require vendors to expose the necessary management functions. An upper-layer controller integrates vendor northbound APIs, delegates configuration, and provides a unified interface to applications. It reuses vendor capabilities for path computation, fault handling, and device adaptation, reducing development effort. These APIs and models are more structured than CLI and can support transactions, access control, and auditing.

Kuiper manages devices through CLI personalization, mapping vendor-specific functions to a unified model and abstract APIs. Operators control the abstraction without requiring vendor controllers or NETCONF/YANG support. This design also accommodates legacy devices and CLI-only deployments.

Our experience implementing vendor-controller orchestration and Kuiper across multiple projects shows two advantages of Kuiper in large, heterogeneous data centers.

Ease of constructing unified abstractions. Vendor controllers differ in abstractions, deployment workflows, and configuration parameters. Reconciling these differences becomes harder as more vendors are added and may limit orchestration to commonly exposed functions. NETCONF/YANG requires

handling differences in model and feature support. Kuiper builds its model around shared protocol definitions and maps each function to vendor-specific CLI. This allows the same control logic to be reused across vendors.

Faster response to changing requirements. Enterprise clouds must adapt to changes in applications and technology. Requirements beyond existing controller APIs or implemented YANG models may require coordinated updates from several vendors. When the required functions are already available through device CLI, we can support them by updating Kuiper's controller logic and CLI adaptations. This reduces cross-vendor coordination and can shorten development cycles.

5.3 Enterprise Cloud Network Orchestration

How to avoid resource conflicts? Large-scale BGP-EVPN deployments struggle to maintain a consistent addressing scheme across hundreds of devices. Traditional manual tracking leads to parameter collisions (e.g., conflicting VNI or VRF IDs) as the scale grows. Kuiper treats these parameters as managed resources, pre-allocating available ranges for L2/L3 VNIs, Bridge Domain IDs, and VRF IDs for each device. Empirical data from deployments with hundreds of VPNs and thousands of Bridge Domains confirms that this design eliminates parameter collision incidents.

How to avoid partial configuration? L3 interconnection and gateway deployment involve a common engineering problem. Minor mismatches in the creation order of L3 VNIs, VRFs, and distributed gateways can leave the network in a partially configured state. Tunnels may be established without routing. We constrained these sequential dependencies using templates. The concrete form of the gateway on each device is selected by Kuiper. As a result, the distributed gateway is no longer manually constructed on a per-device basis but realized as a structured configuration projected from a high-level VPN description to all edge nodes. During multiple cross-region expansions, we did not observe black holes caused by missing gateways on individual nodes. L3 connectivity can be established in a single commit.

Why does Kuiper require verification? Our experience operating Kuiper shows that configuration control for network devices is not binary. A command may be accepted, issued, partially applied, or later drift from the intended state. Kuiper therefore tracks seven states: `Unknown`, `Pending`, `Applied`, `Consistent`, `Inconsistent`, `Deleting`, and `Error`. These states separate control-plane progress from observed device convergence. `Pending` and `Applied` record whether an update has been persisted and issued; `Consistent` and `Inconsistent` record whether the running configuration matches intent; `Deleting` keeps removed objects visible until their absence is verified; `Error` preserves the latest execution failure; and `Unknown` marks objects whose device state has not been revalidated.

This state design is a direct result of failures observed in Kuiper deployments. First, successful CLI execution does not imply configuration convergence. A device may apply a command with delay or reject part of the update. Applied is therefore not a terminal state. Kuiper only promotes an object to `Consistent` after subsequent observation of the device running configuration confirms convergence. Second, deletion in a CLI-driven system is not an atomic transaction. It often spans multiple cycles. Releasing an object immediately after a delete response would cause the controller to lose track of residual configuration on the device. Third, after a restart, previous in-memory judgments may be stale. Until Kuiper re-observes the device, reusing those judgments would turn old control-plane knowledge into false operational fact. The `Unknown` state makes this uncertainty explicit and prevents recovery from relying on unverified configuration.

6 Related Work

Network Control and Management. The evolution of network management has progressed through three phases. First, foundational SDN architectures established the separation of control and data planes [8, 17, 24]. Second, to support hyperscale clouds, systems evolved toward scalable, centralized network virtualization capable of handling millions of tenants [5, 9, 28, 35]. Third, addressing the complexity of diverse infrastructure, the current paradigm emphasizes intent-based networking. Recent studies focus on synthesizing configurations for heterogeneous hardware [2, 3] and automating policy enforcement through high-level abstractions [11, 14, 21, 30].

Configuration Generation. Emerging research [10, 18, 20, 22, 27, 33, 36, 38–40] leverages large language models to synthesize correct network configurations and operational code from high-level requirements. Configuration translation methods [19, 34, 37, 41] focus on bridging cross-vendor interoperability gaps by mapping intents or legacy configurations to target formats.

7 Conclusion

In this paper, we present Kuiper, a plug-and-play network management platform for enterprise cloud networks. Kuiper has been deployed in production for over 6 years across more than 20 enterprises, reducing new-vendor onboarding time from two weeks to a few hours and saving hours of deployment work for application releases. All real-world use cases and questionnaires were anonymized and gathered with the consent of the participants. *This work does not raise any ethical issues.*

Acknowledgments. We sincerely thank our shepherd and the anonymous reviewers for their valuable feedback. This work is supported in part by the National Key R&D Program of China (Grant No. 2024YFB2906800). Han Zhang (zhhan@tsinghua.edu.cn) is the corresponding author.

References

[1] Broadcom. The ai advantage: Running next-gen workloads in private cloud environments. https://news.broadcom.com/cloud/the-ai-advantage-private-cloud-for-next-gen-workloads, 2025. Accessed: 2026-02-01.

[2] Eric Hayden Campbell, William T Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing data plane diversity with control plane synthesis. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 133–153, 2021.

[3] Huangxun Chen, Yukai Miao, Li Chen, Haifeng Sun, Hong Xu, Libin Liu, Gong Zhang, and Wei Wang. Software-defined network assimilation: bridging the last mile towards centralized network configuration management with nassim. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 281–297, 2022.

[4] Container Networking Interface (CNI). Main: interface-creating. https://www.cni.dev/plugins/current/main/. Accessed: 2026-02-01.

[5] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*, pages 373–387, 2018.

[6] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[7] Ashraf Elnashar, Jules White, and Douglas C Schmidt. Prompt engineering for structured data: a comparative evaluation of styles and llm performance. *Artificial Intelligence and Autonomous Systems*, 2(2):32–49, 2025.

[8] Rob Enns, M Bjorklund, J Schoenwaelder, and A Bierman. Rfc 6241: Network configuration protocol (netconf), 2011.

[9] Andrew D Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. Orion: Google's {Software-Defined} networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 83–98, 2021.

[10] Rongxin Han, Jingyu Wang, Qi Qi, Haifeng Sun, Chaowei Xu, Zhaoyang Wan, Zirui Zhuang, Yichuan Yu, and Jianxin Liao. Netren: Service migration-driven network renascence with synthesizing updated configuration. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 708–721, 2024.

[11] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K Reiter, and Vyas Sekar. Intent-driven composition of resource-management sdn applications. In *proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 86–97, 2018.

[12] Brian T. Horowitz. Why some workloads are coming home: The case for cloud repatriation. https://biztechmagazine.com/article/2025/08/why-some-workloads-are-coming-home-case-cloud-repatriation, 2025. Accessed: 2026-02-01.

[13] ITAD Summit. Analyzing hardware refresh cycles in the data center 2020. https://www.itadsummit.com/index.php/itad-blog-analyzing-hardware-refresh-cycles-in-the-data-center-2020/, 2020. Accessed: 2026-02-01.

[14] Arthur S Jacobs, Ricardo J Pfitscher, Rafael H Ribeiro, Ronaldo A Ferreira, Lisandro Z Granville, and Sanjay G Rao. Deploying natural language intents with lumi. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, pages 82–84, 2019.

[15] J.H.I.E. Scrap. It refresh cycle: When to upgrade, replace, or retire hardware. https://jhiescrap.com/it-refresh-cycle-when-to-upgrade-replace-or-retire-hardware/, 2023. Accessed: 2026-02-01.

[16] Richard Johnson. *Network Address Translation Protocols and Design: Definitive Reference for Developers and Engineers*. HiTeX Press, 2025.

[17] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *9th USENIX symposium on operating systems design and implementation (OSDI 10)*, 2010.

[18] Fuliang Li, Haozhi Lang, Jiajie Zhang, Jiaxing Shen, and Xingwei Wang. Preconfig: A pretrained model for automating network configuration. *arXiv preprint arXiv:2403.09369*, 2024.

[19] Chenyang Liu, Fuliang Li, Naigong Zheng, and Xingwei Wang. Netmas: Efficient network configuration translation with multi-agent system. In *2025 IEEE/ACM 33rd International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2025.

[20] Jianmin Liu, Li Chen, Dan Li, and Yukai Miao. CEGS: Configuration example generalizing synthesizer. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 1327–1347, 2025.

[21] Biao Lyu, Enge Song, Tian Pan, Jianyuan Lu, Shize Zhang, Xiaoqing Sun, Lei Gao, Chenxiao Wang, Han Xiao, Yong Pan, et al. POSEIDON: A consolidated virtual network controller that manages millions of tenants via config tree. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1083–1099, 2024.

[22] Sathiya Kumaran Mani, Yajie Zhou, Kevin Hsieh, Santiago Segarra, Trevor Eberl, Eliran Azulai, Ido Frizler, Ranveer Chandra, and Srikanth Kandula. Enhancing network management using code generated by large language models. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 196–204, 2023.

[23] R Marin-Lopez, G Lopez-Millan, and F Pereniguez-Garcia. Rfc 9061: A yang data model for ipsec flow protection based on software-defined networking (sdn), 2021.

[24] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, 38(2):69–74, 2008.

[25] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *Proceeding of IEEE international symposium on a world of wireless, mobile and multimedia networks 2014*, pages 1–6. IEEE, 2014.

[26] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with modeling network topologies at multiple levels of abstraction. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 403–418, 2020.

[27] Rajdeep Mondal, Alan Tang, Ryan Beckett, Todd Millstein, and George Varghese. What do llms need to synthesize correct router configurations? In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 189–195, 2023.

[28] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 426–439, 2016.

[29] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. An analysis and empirical study of container networks. In *IEEE INFOCOM 2018-IEEE Conference On Computer Communications*, pages 189–197. IEEE, 2018.

[30] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. Building scalable and flexible cluster managers using declarative programming. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 827–844, 2020.

[31] Tigera. Configure bgp peering. https://docs.tigera.io/calico/latest/networking/configuring/bgp. Accessed: 2026-04-23.

[32] Tigera. Overlay networking. https://docs.tigera.io/calico/latest/networking/configuring/vxlan-ipip. Accessed: 2026-04-23.

[33] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostić, and Marco Chiesa. Netconfeval: Can llms facilitate network configuration? *Proceedings of the ACM on Networking*, 2(CoNEXT2):1–25, 2024.

[34] Jingyu Wang, Lingqi Guo, Jianyu Wu, Caijun Yan, Haifeng Sun, Lei Zhang, Zirui Zhuang, Qi Qi, and Jianxin Liao. Hierarchical index retrieval-driven wireless network intent translation with llm. *IEEE Transactions on Mobile Computing*, 24(10):9837–9851, 2025.

[35] Chengkun Wei, Xing Li, Ye Yang, Xiaochong Jiang, Tianyu Xu, Bowen Yang, Taotao Wu, Chao Xu, Yilong Lv, Haifeng Gao, et al. Achelous: Enabling programmability, elasticity, and reliability in hyperscale cloud networks. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 769–782, 2023.

[36] Jingtian Wei, Xingrong Gao, Xinyi Cao, Huiqi Deng, Yi Qian, and Peng Zhang. Let's synthesize step-by-step: Generating network configurations with chain of thought. In *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*, pages 1–6, 2025.

[37] Yunze Wei, Xiaohui Xie, Tianshuo Hu, Yiwei Zuo, Xinyi Chen, Kaiwen Chi, and Yong Cui. Inta: Intent-based translation for network configuration with llm agents. In *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*, pages 1–16. IEEE, 2025.

[38] Duo Wu, Xianda Wang, Yaqi Qiao, Zhi Wang, Junchen Jiang, Shuguang Cui, and Fangxin Wang. Netllm: Adapting large language models for networking. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 661–678, 2024.

[39] Xiaofeng Zhang, Xianming Gao, Peilin Tao, and Tao Feng. Graphsynth: Synthesis of network configuration templates using large language models. In *Proceedings of the 2025 2nd International Conference on Generative Artificial Intelligence and Information Security*, pages 108–114, 2025.

[40] Jinyu Zhao, Haifeng Sun, Jingyu Wang, Qi Qi, Zirui Zhuang, Shimin Tao, and Jianxin Liao. Confpilot: A pilot for faster configuration by learning from device manuals. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*, pages 108–119. IEEE, 2023.

[41] Naigong Zheng, Fuliang Li, Ziming Li, Yu Yang, Yimo Hao, Chenyang Liu, and Xingwei Wang. Configtrans: Network configuration translation based on large language models and constraint solving. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*, pages 1–12. IEEE, 2024.

A Configuration Template

By utilizing YAML-based definitions paired with Jinja2-style placeholders, Kuiper decouples high-level intent from low-level CLI execution. As illustrated in the Huawei VRP example, the templates block defines granular command primitives, which serve as reusable building blocks for complex configurations. Each intent path is associated with a unique identity field. This design provides a deterministic way to locate, modify, or delete specific configuration entities. Moreover, Kuiper enables the generation of accurate rollback commands, such as the undo directives defined in the absent field, thereby preventing configuration drift and ensuring that the device state remains synchronized with the central controller. Finally, Kuiper can leverage the statuslets to execute diagnostic commands.

Listing 2: Example of templates for VRFs.

```
vendor: Huawei
os: VRP
version: 5
templates:
  address-family: |
    {{- with .routeDistinguisher }}
      route-distinguisher {{ . }}
    {{- end }}
  ...
  nexthop: '{{ replaces "vbdif" "Vbdif" "null" "NULL" "vlan" "
       Vlanif" . }}'
  ...
  route6: ipv6 route-static vpn-instance {{ .vrf }} {{ toIP .
       network | upper }} {{ cidrPrefixLen .network }}
...
configlets:
  ...
  /{host}/config/v1/vrfs/{name}:
    identity: ip vpn-instance {{ .name }}
    present: |
      ip vpn-instance {{ .name }}
       {{- with .vni }}
       vxlan vni {{ . }}
       {{- end }}
    underlying:
      - identity: vxlan vni *
  /{host}/config/v1/vrfs/{vrf}/ipv6-routes/{network}/nexthops/{
       nexthop}:
    identity: '{{ template "route6" . }} {{ template "nexthop6" .
         nexthop }} *'
    selector: fields(Line, 7) != "public"
    present: '{{ template "route6" . }} {{ template "nexthop6" .
         nexthop }} {{- with .tag }} tag {{ . }}{{ end }}'
    absent: undo {{ template "route6" . }} {{ template "nexthop6" .
         nexthop }}
  /{host}/config/v1/vrfs/{vrf}/ipv6-routes/{network}/nexthops/vrf/{
       nexthop-vrf}:
    identity: '{{ template "route6" . }} vpn-instance {{ .
         nexthopVrf }} *'
    present: '{{ template "route6" . }} vpn-instance {{ .nexthopVrf
         }} {{- with .tag }} tag {{ . }}{{ end }}'
    absent: undo {{ template "route6" . }} vpn-instance
  ...
statuslets:
  /{host}/status/v1/vrfs/{vrf}/ip-routing-table:
    template: display ip routing-table vpn-instance {{ .vrf }}
```

B Current Support for Vendors

We document the validated hardware platforms. Table 6 lists the specific vendors, operating systems, and versions validated in our experience.

Notice: In this paper, we use the term *"new vendor"* in a broad sense. It refers not only to a previously unsupported device manufacturer, but also to a distinct device operating system or system version from an already supported manufacturer. Accordingly, we treat different manufacturers, operating-system families, and system versions as distinct adaptation targets. This definition reflects the fact that command sets often differ across operating systems and versions from the same manufacturer, such as Cisco IOS and Cisco NX-OS.

Table 6: Supported vendors.

Manufacturer	Operating System	Version
Arista	EOS	4
Centec	CentecOS	7
Cisco	NX-OS / IOS	7
DPtech	DPtech OS	1.6
F5	BIG-IP	13
H3C	Comware	7
Hillstone	StoneOS	5
Huawei	VRP	5, 8
Ruijie	RGOS	11
Sino	Sino OS	3
VyOS	VyOS	1.3
ZTE	ZXR10 v2	-
FRR	FRR	8, 9

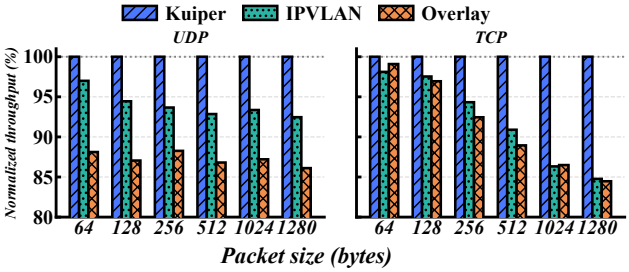


Figure 19: Data plane performance.

C Supplementary Evaluation

Data plane performance under varying packet sizes. Figure 19 depicts the normalized throughput of IPVLAN, an overlay-based solution using VXLAN, and Kuiper. The experiment is conducted by running the benchmark with packet sizes from 64 B to 1280 B under both UDP and TCP, where the throughput of Kuiper is normalized to 100%. By shifting the data plane realization from host-based networking to switch-based hardware VRFs, Kuiper achieves higher throughput than both baselines across the entire range of packet sizes. Averaged over both UDP and TCP, it improves throughput by 7.6% over IPVLAN and 12.8% over the overlay-based solution. The larger gap relative to the overlay-based solution suggests that tunnel encapsulation introduces additional overhead in the data path.